

AFRL-IF-RS-TR-2003-217
Final Technical Report
September 2003



A PUBLISH/SUBSCRIBE BASED ARCHITECTURE OF AN ALERT SERVER TO SUPPORT PRIORITIZED AND PERSISTENT ALERTS

University of Texas at Arlington

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2003-217 has been reviewed and is approved for publication.

APPROVED: /s/
RAYMOND A. LIUZZI
Project Engineer

FOR THE DIRECTOR: /s/
JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 074-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE SEPTEMBER 2003	3. REPORT TYPE AND DATES COVERED Final Jun 01 – Jan 03	
4. TITLE AND SUBTITLE A PUBLISH/SUBSCRIBE BASED ARCHITECTURE OF AN ALERT SERVER TO SUPPORT PRIORITIZED AND PERSISTENT ALERTS			5. FUNDING NUMBERS C - F30602-01-2-0543 PE - 62702F PR - R427 TA - 00 WU - P9	
6. AUTHOR(S) S. Chakravarthy and N. Vontela				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Texas at Arlington 416 Yates Street PO Box 19015 Arlington Texas 76019-0015			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/ITB 525 Brooks Road Rome New York 13441-4505			10. SPONSORING / MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2003-217	
11. SUPPLEMENTARY NOTES AFRL Project Engineer: Raymond A. Liuzzi/ITB/(315) 330-3577/ Raymond.Liuzzi@rl.af.mil				
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.				12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 Words) This report discusses various architectures and implementation issues and discusses our approach for a publish/subscribe based distributed alert server (SPAWAR application) whose requirements include: priority-based delivery, persistence, recovery, time-to-live and various other features. The approach described in this report provides a lightweight implementation that is general-purpose and can be used for a number of applications. A new efficient sweeping algorithm is used to make sure alerts are delivered correctly and satisfy several requirements such as priority, sending existing alerts to new subscribers, and regular expression based subscription. The approach was motivated by a need to provide alert distribution capability based on various needs, such as multiple ways to publish (using TAG, USER, and PROFILE), guaranteed delivery of alerts, asynchronous delivery and ack/receipt distribution. This approach is compared with various alternatives. The sweeping algorithm for priority-based delivery is described in detail.				
14. SUBJECT TERMS Data Knowledge Base, Software, Architecture, Alerting Mechanisms			15. NUMBER OF PAGES 28	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Table of Contents

1. INTRODUCTION	1
1.1. Alternative approaches	2
1.1.1. Client/Server Architecture with remote procedure calls	2
1.1.2. Client/Server Architecture with Object Request Broker	2
1.1.3. Client/Server Architecture with Message Oriented Middleware	3
1.2. Existing Messaging Systems	4
1.3. Motivation.....	4
2. ALERT SERVER OBJECTIVES	4
2.1. Alert Clients	5
2.2. Alert Server.....	5
3. DESIGN OF ALERT SERVER	6
3.1. Alert	6
3.2. Acknowledgement	7
3.3. Receipt	7
3.4. Subscription/Registration.....	8
4. ALERT SERVER ARCHITECTURE	8
4.1. Messaging Domains:.....	8
4.1.1. Point-to-Point Messaging Domain	8
4.1.2. Publish/Subscribe Messaging Domain	9
4.2. Message Consumption:	9
4.3. Message Delivery Mode	10
4.3.1. Logging and Retrieval of Alerts	10
4.4. Subscription and Unsubscription for alerts and acknowledgement.....	11
4.5. Queuing and Distribution of Alerts	12
4.5.1. Queuing	12
4.5.2. Distribution	13
4.5.3. Sweeping Algorithm	13
5. MULTITHREADING THE SERVER	16
5.1. Synchronization Issues	16
5.2. Types of Locks.....	18
5.2.1. Mutex	18
5.2.2. Read-write	18
5.2.3. Semaphore	18
6. CONCLUSIONS AND FUTURE WORK	20
7. REFERENCES	21

List of Figures

Figure 1. Messaging in an enterprise application.	1
Figure 2. Alert Server with alert producer and consumers. C1, C2, ... are clients that can be both producer and consumers of alerts.....	5
Figure 3. Contents of a log file. LSN is the log sequence number, fp is the file position and size is the size of the object.....	11
Figure 4. Hash tables that store the consumers that are registered with alert server for a topic. All consumers registered for an alert are stored in ALT and those registered for an acknowledgement are stored in ACK table. A, B, C are the topics and C1, C2, C3... are the consumers.	12
Figure 5. Priority queue data structure. Each alert is stored in its respective priority level queue. A1, A2... are alerts with their Ids that are stored in this data structure. ...	13
Figure 6. Sweeping algorithm that sweeps through the priority queue.	15
Figure 7. Alert Server Architecture Overview.....	17

List of Tables

Table 1. Locks used for DIFFERENT DATASTRUCTURES	19
--	----

A PUBLISH/SUBSCRIBE BASED ARCHITECTURE OF AN ALERT SERVER TO SUPPORT PRIORITIZED AND PERSISTENT ALERTS

ABSTRACT

This paper discusses various architectures and implementation issues and discusses our approach for a publish/subscribe based distributed alert server (a SPAWAR application) whose requirements include: priority-based delivery, persistence, recovery, time-to-live and various other features. The approach described in this paper provides a lightweight implementation that is general-purpose and can be used for a number of applications.

A new efficient sweeping algorithm is used to make sure alerts are delivered correctly and satisfy several requirements such as priority, sending existing alerts to new subscribers, and regular expression based subscription. Our approach was motivated by a need to provide alert distribution capability based on various needs, such as multiple ways to publish (using TAG, USER, and PROFILE), guaranteed delivery of alerts, asynchronous delivery and ack/receipt distribution. Our approach is compared with various alternatives. The sweeping algorithm for priority-based delivery is described in detail.

1. INTRODUCTION

Enterprise messaging products (or as they are sometimes called, Message Oriented Middleware products or MOM) are becoming an essential component for integrating intra-company operations. They allow separate business components to be combined into a reliable, yet flexible, system. In addition to the traditional MOM vendors, several database vendors and a number of Internet related companies also provide enterprise-messaging products.

MOMs [4] have a number of applications. For example, components of an enterprise application for an automobile manufacturer can use the Alert Server in situations where:

- The inventory component can send a message to the factory component when the inventory level for a product goes below a certain level, so the factory can make more cars.
- The factory component can send a message to the parts components so that the factory can assemble the parts it needs.
- The parts components in turn can send messages to their own inventory and order components to update their inventories and order new parts from suppliers.
- Both the factory and parts components can send messages to the accounting component to update their budget numbers.
- The business can publish updated catalog items to its sales force.

Figure 1. Using messaging for these tasks allow the different components to interact with each other efficiently, without tying up network or other resources. Manufacturing is only one example of how an enterprise can use messaging system and it's API. This can also be used in financial services applications; health services applications and many more applications. Thus the basic aim in all these applications is the distribution of processing across multiple processors and platforms.

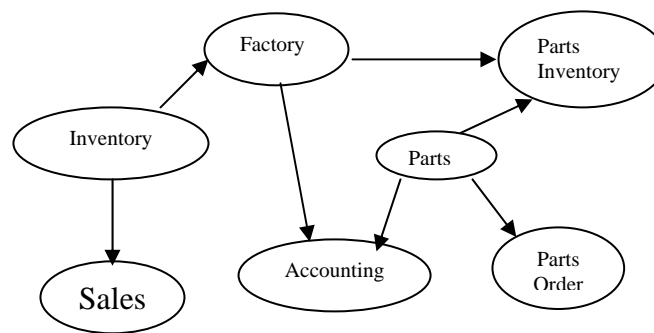


Figure 1. Messaging in an enterprise application.

1.1. Alternative approaches

1.1.1. Client/Server Architecture with remote procedure calls

In this architecture, different applications try to communicate by making remote procedure calls (RPC) in which the messages are sent. Because they are embedded, RPCs [8] do not stand alone as a discreet middleware layer. When the client program is compiled, the compiler creates a local stub for the client portion and another stub for the server portion of the application. These stubs are invoked when the application requires a remote function and typically support synchronous calls between clients and servers. Thus this architecture reduces the complexity of developing applications that span multiple operating systems and network protocols by insulating the application developer from the details of the various operating system and network interfaces [1]. However, RPC is appropriate for client/server applications in which the client can issue a request and wait for the server's response before continuing its own processing. Because most RPC implementations do not support peer-to-peer, or asynchronous, client/server interaction, RPC is not well suited for applications involving distributed objects or object-oriented programming.

Asynchronous and synchronous mechanisms each have strengths and weaknesses that should be considered when designing any specific application. In contrast to asynchronous mechanisms employed by Message-Oriented Middleware, the use of a synchronous request-reply mechanism in RPC requires that the client and server are always available and functioning (i.e., the client or server is not blocked). In order to allow a client/server application to recover from a blocked condition, an implementation of a RPC is required to provide mechanisms such as error messages, request timers, retransmissions, or redirection to an alternate server. The complexity of the application using a RPC is dependent on the sophistication of the specific RPC implementation (i.e., the more sophisticated the recovery mechanisms supported by RPC, the less complex the application utilizing the RPC is required to be). RPCs that implement asynchronous mechanisms are very few and are difficult (complex) to implement.

1.1.2. Client/Server Architecture with Object Request Broker

An object request broker (ORB) is a middleware technology that manages communication and data exchange between objects. ORBs promote interoperability of distributed object systems because they enable users to build systems by piecing together objects – from different vendors – that communicate with each other via the ORB. The implementation details of the ORB are generally not important to developers building distributed systems. The developers are only concerned with the object interface details. This form of information hiding enhances system maintainability since the object communication details are hidden from the developers and isolated in the ORB. The two major ORB technologies are Object Management Group's (OMG) Common Object Request Broker Architecture specification and Microsoft's Common Object Model. Even though CORBA [2] has certain advantages, it does not support the transfer of objects. There is also no garbage collection [5]. Moreover ORBs developed by different vendors may have significantly different features and capabilities. Thus, users must learn a

particular vendor specification, and their value-added features that are often necessary to make a CORBA product usable. Similarly COM/DCOM [7] has certain negative aspects that make it unsuitable in implementing such systems. It is platform specific and there is no stability of APIs making maintainability of software difficult in the long run.

1.1.3. Client/Server Architecture with Message Oriented Middleware

Message-oriented middleware (MOM) is a client/server <http://www.sei.cmu.edu/str/descriptions/clientserver.html> infrastructure that increases the interoperability, portability, and flexibility of an application by allowing the application to be distributed over multiple heterogeneous platforms. It reduces the complexity of developing applications that span multiple operating systems and network protocols by insulating the application developer from the details of the various operating system and network interfaces [1]. MOMs generally support synchronous calls between clients and server applications [4]. Message queues provide temporary storage when the destination program is busy or not connected. MOM reduces the involvement of application developers with the complexity of the master-slave nature of the client/server mechanism. MOM increases the flexibility of architecture by enabling applications to exchange messages with other programs without having to know what platform or processor the other application resides on within the network. The aforementioned messages can contain formatted data, requests for action, or both. MOM is typically asynchronous and peer-to-peer, but most implementations support synchronous message passing as well. MOM is typically implemented as a proprietary product, which means MOM implementations are likely to be incompatible with other MOM implementations. Using a single implementation of a MOM in a system will most likely result in a dependence on the MOM vendor for maintenance support and future enhancements. This could have a highly negative impact on a system's flexibility, maintainability, portability, and interoperability. It is for these reasons that the Alert Server is built using the queuing mechanism that MOMs use and Java RMI for the distribution of messages between different autonomous, heterogeneous processes.

This paper discusses the design and implementation of one such messaging system called the Alert Server. Alert Server is a general-purpose alert and acknowledgement message queue and distribution mechanism. It maintains transaction logs for a comprehensive audit trail of alerts, acknowledgements and receipts. In general, an alert is generated and submitted to the Alerts Server based upon some mission application criteria or condition. At the alerts server, the alerts are logged and queued and if necessary persisted. The Alerts Server determines if there are any subscribers for this alert and if so, forwards it to the destination. An alert producer could be a human operator who "fills in the blanks" of an alert message through a GUI or other means. Alert producers can also be software components that execute "under the hood", invisible to human operators. An alert producer assembles the alert in reaction to some system condition and then sends for distribution. Alert consumers are those applications that are interested in receiving (a subset of) alerts. This is always accomplished via "registering" or "subscribing" for alerts that contain a particular pattern in the alert destination or topic data element by specifying a filter (in the form of a regular expression) during alert registration. Any client application can be either an alert producer or a consumer or both

as long as they use the alert server APIs to communicate with the alert server. This Server has been designed to handle C/C++ as well as Java clients.

1.2. Existing Messaging Systems

Messaging systems are peer-to-peer facilities. In general, each client can send messages to, and receive messages from any client. Each client connects to a messaging agent, which provides facilities for creating, sending and receiving messages. Each system provides a way of addressing messages. Each provides a way to create a message and fill it with data. Some systems are capable of broadcasting a message to many destinations. Others only support sending a message to a single destination. Some systems provide facilities for asynchronous receipt of messages (messages are delivered to a client as they arrive). Others support only synchronous receipt (a client must request each message). Each messaging system typically provides a range of services that can be selected on a per message basis. One important attribute is the lengths to which the system will go to insure delivery. This varies from simple best effort to guaranteed to only once delivery. Other important attributes are message time-to-live, priority and whether a response is required. There are many messaging systems that are being used. IBM's MQSeries, Sun's Java Message Queue and Sentinel's GED are only some of the few. GED is a distributed event based system that supports event detection across address spaces, provide interface for event registration and notification, and support composite event detection. Current version of GED [9] does not guarantee the delivery of events and does not implement the publish/subscribe paradigm. It does not support delivery of events on the content based filtering.

1.3. Motivation

This paper discusses the design and implementation of a messaging system called the Alert Server. There are several motivations behind our objective of designing and implementing a message distribution mechanism even though we have many existing messaging systems. First, not all MOM implementations support all operating systems and protocols. Second, internal infrastructure of these MOMs cannot be modified to achieve our goals. Third, there is high overhead if we build the Alert Server on top of existing MOMs. Finally, some of the special requirements, such as time-to-live, persistence, and choice of receipt/acknowledgment cannot be easily handled by available systems.

2. ALERT SERVER OBJECTIVES

If our Alert Server provided a union of all the existing features of messaging systems it would be much too complicated for its intended users. It is crucial that the Alert Server includes appropriate functionality needed to implement sophisticated enterprise applications.

Our design and implementation of the alert server attempts to minimize the set of concepts a programmer must learn to use enterprise-messaging products. It strives to maximize portability. We start with the concept of alert producers, consumers and distributors that act as servers.

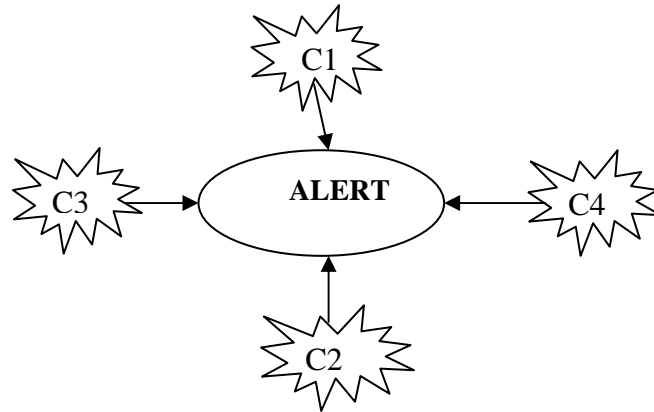


Figure 2. Alert Server with alert producer and consumers. C1, C2, ... are clients that can be both producer and consumers of alerts.

2.1. Alert Clients

A client can be either a producer or a consumer or both. The alert producer does not necessarily need to know who the receiver(s) of the message will be. The producer “publishes/sends” the messages to the Alert Server which is responsible for the distribution of messages. Alert Consumers are responsible for processing and responding to the alert (message) by subscribing/registering through the alert server. An important goal in this case is to minimize the work needed to implement a producer or a consumer.

2.2. Alert Server

Alert Server manages the alert and acknowledges message queues, distribution of alerts, and crash recovery. Important goals of an alert server are:

1. Implement a publish/subscribe model. This model has been chosen over the point-to-point model because point-to-point (PTP) models are built around the concept of message queues. Each message is addressed to a specific queue; clients extract messages from the queue(s) established to hold their messages. Clients have to pull the message from the server rather than the server pushing it to the client after processing the messages. Publish and subscribe (Pub/Sub) clients send messages to the alert server. Publishers and subscribers are generally anonymous and may dynamically publish or subscribe to the alert server. The alert server pushes the messages arriving from a node’s multiple publishers to its (multiple) subscribers.
2. Insurance of delivery of alerts before it expires (time-to-live) based on priority.
3. Dynamic delivery of alerts between multiple producers and consumers based on their registration/subscription topics.
4. Persistence of alerts and acknowledgements to handle to crash recovery of clients.
5. Maintain the privacy and integrity of the messages.

3. DESIGN OF ALERT SERVER

This section discusses the design of the Alert Server. First, we describe the functionality to be supported by the Alert Server and then describe how the Alert Server has been designed to achieve this functionality. It also discusses the design of a proxy server, which acts as a proxy to the Alert Server, to enable the C/C++ clients in communicating with the Alert Server. This section also discusses the assumptions made in designing the proxy server.

The Alert Server should provide API to send messages from one application to another. The client applications should also be able to register and unregister topics of interest. They should be able to cancel messages and should also be able to send acknowledgements and/or receipts. Besides, the Alert Server should have the delivery logic in order to send, and if necessary persist, messages to different destinations on the basis of their priority. Finally, the alert server should be able to recover from crashes. The next section describes different types of messages handled by the alert server.

3.1. Alert

An alert message contains an alert header as well as an optional alert body. All messages support the same set of header fields. Header fields contain values used by both clients and the Alert Server to identify and route messages. Body, on the other hand, can be any Java Object for Java-based clients and an arbitrary string for C/C++ clients. A header of the alert message has the following data elements.

1. Destination: The destination field in the data element is the “topic” and synonymous to, for example, a message “topic” or an email “subject”. It is the value in this field the alert consumers register or unregister by specifying a pattern in the registration request. The value of this field must begin with one of 3 prefixes (with colon included and all the letters capitalized) TAG:, PROFILE:, USER:
2. Alert Type: This field in the header identifies the message type. There are basically two types of alerts, alert itself and an acknowledgement for the alert. An alert is specified by setting the alert type data element value to Alert, where as an acknowledgement is specified by setting the alert type to Acknowledgement.
3. Duration: Duration data element in the header helps in identifying the messages that have expired. Alerts or acknowledgements will remain active and available for distribution from the Alert Server according to its time-to-live indicator. Once the alert server receives and forwards the message to any registered recipients, it will remain in the Server’s queue for the specific duration. The Server deletes the expired messages. Indefinite storage of messages in the queues is handled by setting this data field to zero. These alerts should be explicitly cancelled by the original producer or by any client.
4. Priority: This field in the header helps in priority based delivery of messages. The priority levels are 0-9, where 0 is designated as the lowest and 9 as the highest priority. Messages having the same priority are delivered in the order they arrive (first come first serve basis).

5. Classification: This part of header information allows application specific classification. Unclassified, Confidential, Secret and Top Secret are the four classification levels.
6. Persistence: The producer designates messages as persistent, by setting this field in the header. The Alerts Server stores persistent messages so that they are reloaded in case of restart and recovered in the case of crash. The storing of the messages and their retrieval is discussed later.
7. Acknowledge Policy: This field ensures the delivery of messages to the destination. An alert can have one of the three acknowledgement policies attached to it: None, Client Acknowledgement, and Receipt. A client acknowledgement requires the receiving client to generate an acknowledgement alert where as a client receipt is automatically constructed and submitted to the server after the client is notified of an incoming alert. Client receipts are not stored as clients make blocking calls when they send alerts that require receipts. Also Alert Server discards acknowledgement for alerts that do not require acknowledgement.
8. Cancel Policy: This field helps in the cancellation of alerts that have lived indefinitely on the server. An alert can be cancelled i.e., deleted from the queue on the server, by any client if the Cancel Policy field is set to ANY, or only by the producer of the alert if it is set to the ORIGINATOR.
9. Alert ID: Alert ID is a unique integer that is generated by the alert server to identify a particular alert or acknowledgment.
10. Correlation ID: This data element is only used when the message is an acknowledgment. This field takes on the value of the Alert ID for which it is an acknowledgment.
11. Body: The alert body is any JAVA object that can be sent with the message while it is a character string in the case of C/C++ alert producers. This difference is due to the limitation of C and C++. The body in the case of acknowledgement is a string "ACK".

3.2. Acknowledgement

An alert acknowledgement as explained above is actually an alert. It contains an alert header with the same destination or topic as its associated alert, origination time, persistence, time-to-live (duration) etc. The alert body for the Java based API is replaced by the string "ACK". The correlation ID data element contains the alert ID value of the alert to which this acknowledgment message is responding. Alert consumer applications may register just for alert acknowledgements, without having to register for the alert. This capability is provided so that application developers can create a chain of events, for example: A to produce an alert, B to register for this alert, C to register for acknowledgements for this alert. A sends alert, B receives alert and acknowledges it, C receives acknowledgement.

3.3. Receipt

An alert producer can request a "delivery" receipt for an alert by setting alert's acknowledgement policy data field to Client Receipt. The receipt is automatically

constructed by the Alert Server the instant the alert consumer receives the alert and then forwarded by the Alert Server to the requesting producer. Unlike acknowledgements, receipts are not queued or stored at the Server, nor can alert consumers register for receipts. This feature is simply to ensure the delivery of the alert.

3.4. Subscription/Registration

As explained above every alert contains a destination header field, the value of which starts with one of the prefixes: For example: TAG: Alert, PROFILE: watch officer, USER: Smith. The TAG: prefix helps alert consumers to subscribe to receive specific alerts by specifying a filtering mechanism that employs regular expression masks. For example: .* specifies all alerts. ABC specifies all alerts where destination contains the string ABC. ^A specifies all alerts where destination begins with A. X\$ specifies all alerts where destination ends with X. PROFILE: prefix helps alert consumers to subscribe to any message that belong to a specific profile. The USER: prefix as the name indicates helps in subscribing to a messages to a particular user.

When the Alert Server processes an incoming alert, it places the filtering mask that the consumers subscribed upon the pattern contained in the alert's destination. For example, if there is a consumer subscribed to a filter of "TAG:Alert" and an alert with the destination "TAG:AlertXYZ" comes in, then the subscription "TAG:Alert" is matches against the topic or destination "TAG:AlertXYZ" because the regular expression mask of "TAG: alert" placed against the topic "TAG: alertXYZ" matches true ("TAG: alert is a sub string of "TAG:AlertXYZ"). In the case of the other prefixes, the matching is performed by string comparing the subscription filter with the destination in the alert header. Consumer clients have the option to subscribe to multiple topics using separators. Multiple topics with the same prefix are submitted by separating with commas, for example, TAG:a, b, c will produce three subscriptions; TAG:a, TAG:b, and TAG:c. Similarly, multiple topics with multiple prefixes are submitted using a semicolon to separate prefixes, for example, TAG:a, b;USER:a, b will produce four subscriptions, TAG:a, TAG:b, USER:a, USER:b. The next section discusses the alert server architecture. It tries to explain the reasons for different decisions taken for designing the architecture.

4. ALERT SERVER ARCHITECTURE

4.1. Messaging Domains:

Most messaging products support either point-to-point or the publish/subscribe approach to messaging.

4.1.1. Point-to-Point Messaging Domain

A point-to-point (PTP) product or application is built around the concept of message queues, senders, and receivers. Each message is addressed to a specific queue, and receiving clients extract messages from the queue(s) established to hold their messages. Queues retain all messages sent to them until the messages are consumed or until the messages expire.

PTP messaging has the following characteristics:

- Each message has only one consumer.
- There are no timing dependencies between a sender and a receiver of a message. The receiver can fetch the message whether or not it was running when the client sent the message.
- The receiver acknowledges the successful processing of a message.

This messaging model is useful when the message needs to be processed successfully by one consumer. But, as already mentioned, one of the objective of Alert Server is the delivery of messages to multiple clients. Therefore the architecture of Alert Server implements a publish-subscribe model that is explained below.

4.1.2. Publish/Subscribe Messaging Domain

In a publish/subscribe (pub/sub) product or application, clients address messages to a topic. Publishers and subscribers are generally anonymous and may dynamically publish or subscribe to the content hierarchy. The system takes care of distributing the messages arriving from a topic's multiple publishers to its multiple subscribers.

Pub/sub messaging has the following characteristics:

- Each message may have multiple consumers.
- There is a timing dependency between publishers and subscribers, because a client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.

The Alert Server uses this messaging model as it has to delivery messages to zero, one or many consumers that are anonymous. This timing dependency is relaxed by allowing the producers to create persistent alerts. Persistent alerts can be received even when the subscribers are not active. Thus, persistent alerts provide the durability and reliability provided by the queues and still allows clients to send alerts to many consumers.

4.2. Message Consumption:

Messages can be consumed in either of the two ways.

- Synchronously: A subscriber or a receiver explicitly fetches the message from the destination by calling a method. The method can block until a message arrives, or it can time out if a message does not arrive within a specified time limit.
- Asynchronously: A subscriber need not wait for the delivery of the message. Whenever the message arrives, the server forwards it to the consumers that have registered for that message. The consumers do not have to wait for the delivery of the message.

4.3. Message Delivery Mode

The producers send alerts to the Alert Server in two modes. As already explained the producers can set the delivery mode of the message in the persistent header field.

- The NON-PERSISTENT mode is the lowest-overhead delivery mode because it does not require that the message be logged to stable storage. Alert Server failure can cause a NON-PERSISTENT message to be lost.
- The PERSISTENT mode instructs the Alert Server to take extra care to insure the message is not lost in transit due to its failure. It logs the alerts and helps in retrieving them during normal start ups as well as in the case of recovery after crashing. The data structures used in the fast retrieval of alerts is explained in the next section.

4.3.1. Logging and Retrieval of Alerts

The persistent mode delivery of the alerts causes them to be stored to a stable storage (disk). The alerts are stored in files depending on their priority. There is a file for each level of priority. Each file contains the following:

1. Index Table: The index table is a data structure stored in the log file for fast retrieval of alerts. Each log file has its own index table to store and retrieve the alerts belonging to its priority. The index table reduces the search time and thus helps in the fast retrieval of alerts. The index table and the alerts are stored in a serialized manner. The table has records that hold the information of the location of the alert in the log. Each record has a log sequence number (LSN), pointer to the position of storage of the alert in the file (fp), size of the alert (size) and a cancel bit. The log sequence number helps in indexing into the table and obtain the record that has information regarding the storage of alert. The record has file pointer that points to the alert in the file and the size of the file.
2. BLSN: BLSN in each file is an integer stored in each log along with the index table. BLSN is set to the ID of the alert that has recently been added to the priority queue. There is a BLSN for each priority queue in its corresponding log.
3. DLSN: DLSN, like BLSN, is also an integer stored in each log for corresponding priority queue. DLSN, unlike BLSN, is set to the ID of the alert that has been recently sent from the priority queue.
4. CANCEL BIT: The cancel bit indicates whether the alert has been cancelled or not.

Both DLSN and BLSN help in the retrieval of NON-PERSISTENT alerts in case of crash of the Alert Server. Instead of reading all the non-persistent alerts from the log when the server recovers, only those alerts whose IDs fall between BLSN and DLSN are read from the log and put into the priority queue, thus reducing the number of alerts read from the logs. The serialized log files always contain capacity of the index table at beginning followed by BLSN after 4L bytes. DLSN is at 8L bytes followed by index

table with its records at 12L bytes, followed by storage of actual alerts from 20L bytes in each file. An example log file for one of the priority levels is shown in Figure 3.

The alerts, after storing in the log, are sent to the queue for distribution to the consumers who have subscribed to their topics. Since acknowledgements are alerts with just the correlation ID set to that of their alert, they are handled as if they are alerts.

1. Capacity of the log (0L)
2. BLSN (4L)
3. DLSN (8L)
4. Index Table (12L)

LSN	fp	Size
1	20L	24L
3	45L	27L

Alert with ID 1
Alert with ID 3

Figure 3. Contents of a log file. LSN is the log sequence number, fp is the file position and size is the size of the object.

4.4. Subscription and Unsubscription for alerts and acknowledgement

Consumers that have subscribed for specific alerts or acknowledgements are stored in hash tables. There are three primary hash tables one for each prefix TAG, USER and PROFILE. These hash tables have entries with alert type as key and another secondary hash table as value. The secondary hash table has keys for each topic, pointing to the consumer list as their values. This list contains consumer nodes that hold information regarding different consumers registered with the Alert Server. Each node in the list has unique ID. Apart from this, the list also contains information regarding the number of consumers added and number of consumers deleted. For each consumer added to the list, the attribute of the list increases by one while each deletion decrements the other attribute by one. These attributes help in reducing the sweep time of the priority queue. This is explained later in detail. Every consumer list in the secondary hash table has a unique ID. All consumers registered for alerts are stored in one hash table while all consumers are registered for ACKs are stored in another secondary hash table. Consumers for a specific topic are stored in the same consumer list in the secondary hash table. All consumers for an alert are obtained by using the alert topic as the key in the secondary hash table. Every new consumer registration causes that consumer object to be added to the beginning of its respective list. The primary hash table at the top level is for indexing purpose. This reduces the search time in finding the consumers registered for a specific alert type (ALERT or an ACK). The second level hash table is used for

determining the consumers for a specific topic. An hash table has been chosen for the first level over other possibilities of array of lists as it would be easier to add more keys, such as CLIENT RECEIPT if needed in the future. The unsubscription for alerts and acknowledgements by a consumer results in the deletion of that consumer from the consumer list preventing the Alert Server from distributing any messages further.

The data structure with the hash table and consumer lists is shown in Figure 4. The primary hash table contains topics as keys while the secondary hash table distinguishes the consumers registered for alerts and acknowledgements.

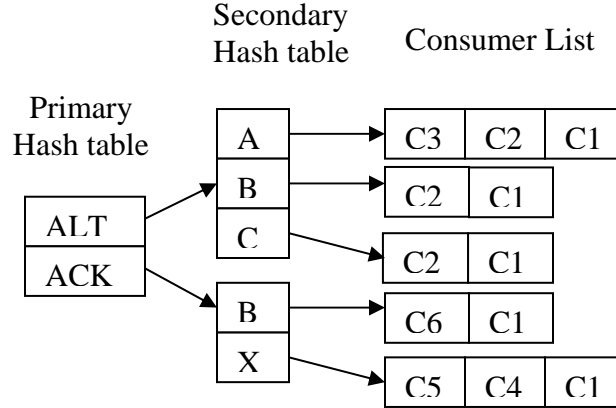


Figure 4. Hash tables that store the consumers that are registered with alert server for a topic. All consumers registered for an alert are stored in ALT and those registered for an acknowledgement are stored in ACK table. A, B, C are the topics and C1, C2, C3... are the consumers.

4.5. Queuing and Distribution of Alerts

The alerts are stored in the logs if needed depending on their delivery mode and later put in the priority queue. There are ten queues one for each priority level. The data structure used for the queues is an array of queues. The alert priority level is used for indexing into the array and getting the queue at that index. Every new alert is always added at the beginning of the queue. Therefore insertion of the alert always takes a constant time. Queuing and distribution of alerts are two independent operations. Therefore they are handled simultaneously using two different threads. The data structures used in queuing and the algorithm for sweeping are explained below.

4.5.1. Queuing

Producers publish alerts independent of the distribution of alerts on the Alert Server. Therefore they are queued for delivery by the Alert Server. The queuing of alerts is simple. Whenever a new alert comes in, it is indexed into the queues array using its priority and then put in the beginning of its queue. After putting the alert into the queue, the BLSN of the log file depending on the alert priority level is updated with the id of the alert. Thus the alerts put into the queue are now available for distribution. The queue is

swept in order to distribute the alerts. Similarly, when an alert is cancelled, the alert is removed from the queue. The canceling of the alert depends on its cancellation policy. The expired alerts are removed when the queue is swept. The queue array with queues for each priority level is shown below in Figure 5. The priority of alert is used as an index into this array of queues that will reduce the time in searching and adding an alert at its proper position.

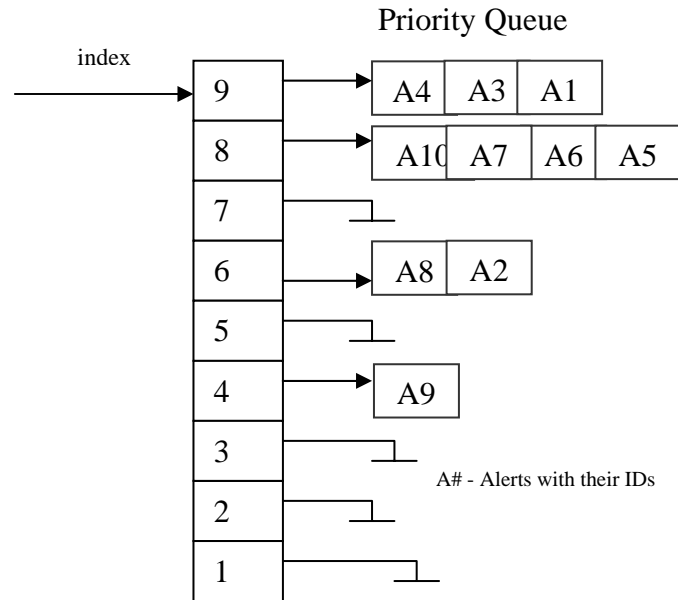


Figure 5. Priority queue data structure. Each alert is stored in its respective priority level queue. A1, A2... are alerts with their IDs that are stored in this data structure.

4.5.2. Distribution

Alerts are distributed to their respective consumers by comparing their topic with the topic in the consumer object that is created and stored in the consumer list data structure when the consumer registers. The priority queues are swept and the alerts are distributed. It is during this sweeping of the priority queue that the expired alerts are purged. Since the goal of distribution is the delivery of alerts on the basis of priority, the higher priority level alerts are delivered before the lower ones. Higher priority numbers indicate higher priority. Alerts of the same priority are delivered on LCFS (last come first served) basis since the new alerts are added at the beginning of the queue. The way in which the queues are swept is discussed next.

4.5.3. Sweeping Algorithm

The algorithm makes use of the information held in the alert and consumer objects. The alert object, apart from the header and body field, contains a hash table that holds mapping between each consumer list and max ID of the consumer in the list that it has been sent. This hash table contains the consumer list id as the key and the highest consumer ID that received the alert as its value. This information is necessary in trying to

stop sending the message to consumers that have already received it. This also reduces the time for sweeping the priority queues. As already explained, each consumer object in the consumer list has unique ID in that list and similarly every consumer list also has its ID. Consumer list ID serves as an index to the hash table of the alert message which indicates that the alert has been sent up to this consumer in the list and needs to be sent to all the consumers before this consumer in the list. Since new consumers are always added at the beginning of the list, the consumers are always in decreasing order of their ID. This information is used to prevent from resending the same message to the same consumer more than once. The sweeping algorithm guarantees the delivery of alerts in the order of their priority. The algorithm takes a queue of the highest priority from the priority queue data structure and then traverses the queue to send the alerts to the registered consumers. There are two possibilities of changes to the priority queue and consumer table data structures: either an alert has been added or deleted from the one of the queues resulting in change of state of the priority queue data structure or some consumers might have been added or deleted from the consumer list resulting in a change in consumer table.

The algorithm executes in three phases. In the first phase the alert is checked for expiration. Expired alerts are removed from the queue. This is checked by comparing current system time with the sum of the time at which alert was received on the server and the time-to-live data field in the alert header. If the sum is greater than the system time then it is removed from the queue. Phase two consists of finding the consumer list. Comparing the alert destination with the keys in the consumer hash tables does this. This phase also filters out consumers that have been registered for alerts that may be published in future. The consumer lists thus obtained are used further in phase three. In the absence of a consumer list, the sweeping algorithm continues with the next alert in the queue and applies the same three phases. During the third phase, the hash table in the alert is used to reduce the traversal of the consumer list. This also prevents resending of alerts to the same consumers. The unsubscription of consumers is not of much concern as they are simply deleted from the consumer list and there is no need to worry about the delivery of the alerts to them. During the third phase, the max ID of the consumer the alert had been sent to is obtained from the hash table in the alert by using the consumer list id which serves as a key. If this is zero, it means we have a new alert in the queue and the alert is sent to all the consumers in the list. If the ID of the consumer retrieved is more than zero, then the alert has been sent to some consumers and needs to be sent to the newly added ones.

Algorithm: Sweeping algorithm is shown in Figure 6. Initially, queue Q with the highest priority is sent to the algorithm. Let $A_0, A_1, A_2 \dots$ be the alerts in the queue. Their subscripts indicate their positions. Alerts are added at the beginning of the queue. $A_{current}$ be the current alert that is being distributed in the queue and A_{next} be the alert after the current alert. ST and PT are the secondary and primary hash tables that store the information about the consumers registered. The descriptions of these data structures have been described in earlier sections. HT is another hash table in each alert. This stores the consumer ID that has recently received the alert in that consumer list. All the lists that have the same topic that match the alert topic have an entry in this table. $X[Y]$ indicates an attribute X of an object Y. For example, topic $[A_{current}]$ indicates the topic field of the alert.

```

SweepingAlgorithm (Q) {
    1.  $A_{\text{current}} = A_0$ 
    2. while ( $A_{\text{current}} \neq \text{null}$ ) begin
    3.   if ( $\text{expired}(A_{\text{current}}) = \text{false}$ )
    4.     findconsumerlist( $A_{\text{current}}$ )
    5.   else
    6.      $Q = Q - A_{\text{current}}$ 
    7.      $A_{\text{current}} = A_{\text{next}}$ 
    8.   end of while

```

Figure 6. Sweeping algorithm that sweeps through the priority queue.

```

findconsumerlist ( $A_{\text{current}}$ )
    1.  $ST = PT(\text{alerttype}[A_{\text{current}}])$ 
    2. if ( $\text{prefix} = \text{TAG}$ )
    3.   for each key in  $ST$  begin
    4.     if ( $\text{matches}(\text{key}, \text{topic}[A_{\text{current}}])$ )
    5.        $\text{list} = ST[\text{topic}[A_{\text{current}}]]$ 
    6.        $\text{sendtoOutputQueue}(\text{list}, A_{\text{current}})$ 
    7.   end
    8. else
    9.    $\text{list} = ST[\text{topic}[A_{\text{current}}]]$ 
    10.   $\text{sendtoOutputQueue}(\text{list}, A_{\text{current}})$ 

    sendtoOutputQueue ( $\text{list}, A_{\text{current}}$ )
    1. if ( $\text{list} \neq \text{null}$ )
    2.   $\text{consrecv} = \text{get}(\text{HT}[\text{ID}[\text{list}]] [A_{\text{current}}])$ 
    3.   if ( $\text{consrecv} = 0$ )
    4.     send to all the consumers between  $\text{added}[\text{list}]$  and  $\text{deleted}[\text{list}]$ 
    5.     put ( $\text{HT}[A_{\text{current}}], \text{added}[\text{list}]$ )
    6.   else
    7.     if ( $\text{added}[\text{list}] > \text{consrecv}$ ) {
    8.       send to all consumers between  $\text{added}[\text{list}]$  and  $\text{consrecv}$ 
    9.       put ( $\text{HT}[A_{\text{current}}], \text{added}[\text{list}]$ )

```

5. MULTITHREADING THE SERVER

The Alert Server uses Java Remote Method Invocation [RMI][3] in its communication interface. RMI calls are blocking therefore these calls need to be handled asynchronously. Client requests are queued. Since each request is independent and there is no guarantee that they will arrive within a certain time there is a queue for each type of request and a different thread handles each different request. Multithreading also helps in making the server scalable.

The clients put the messages in the queue and continue with their processing. Since each queue has a thread listening on it, the thread is awakened when the queue is not empty. The data structures handled by each thread are shown in Figure 7. There are other threads for handling other requests like canceling an alert, unsubscribing for a topic. The threads shown in Figure 7 are the threads that handle registration for a topic, publishing an alert and the delivery of alerts to different consumers. The publish handler listens on the notify buffer that holds the alerts that are published by different clients. It places these alerts in the priority queue data structure on the basis of their priorities. On the other hand, the registration thread handles the registration in the registration buffer independent of publishes. This thread constructs consumer nodes that hold consumer information that is used while sending the alert to the respective consumers by the output handler thread and places them in their respective consumer lists in the secondary consumer hash table. The Message handler thread runs the sweeping algorithm on the priority queues and places the alert and its consumers in the output queue. The output handler thread picks up these alerts and delivers them to the consumers. This thread makes RMI calls to the clients to deliver the messages.

5.1. Synchronization Issues

The Alert Server is made up of several data structures that will be shared and hence may be concurrently accessed by threads. Following is the list of shared data structures:

1. Consumer list: list of consumers with their IP addresses and their USER ID.
2. Priority Queues: array of ten queues that store the alerts on the basis of their priority.
3. Output Queue: queue of the alerts and the consumers registered.
4. Alert log file: There is one log file for each priority. It stores the alerts and retrieval information.
5. Secondary Hash table: hash table that helps to distinguish between different topics for the same alert types. There is a hash table for each alert type ALERT and ACK.

Race Conditions: When the result of two or more threads performing an operation depends on unpredictable timing factors, there is a race condition. Hence the access of the consumer list and several such shared data structures must be guarded for mutual exclusion. This can be attained using synchronization mechanisms or locks. There are several types of locks and the right choice must be made.

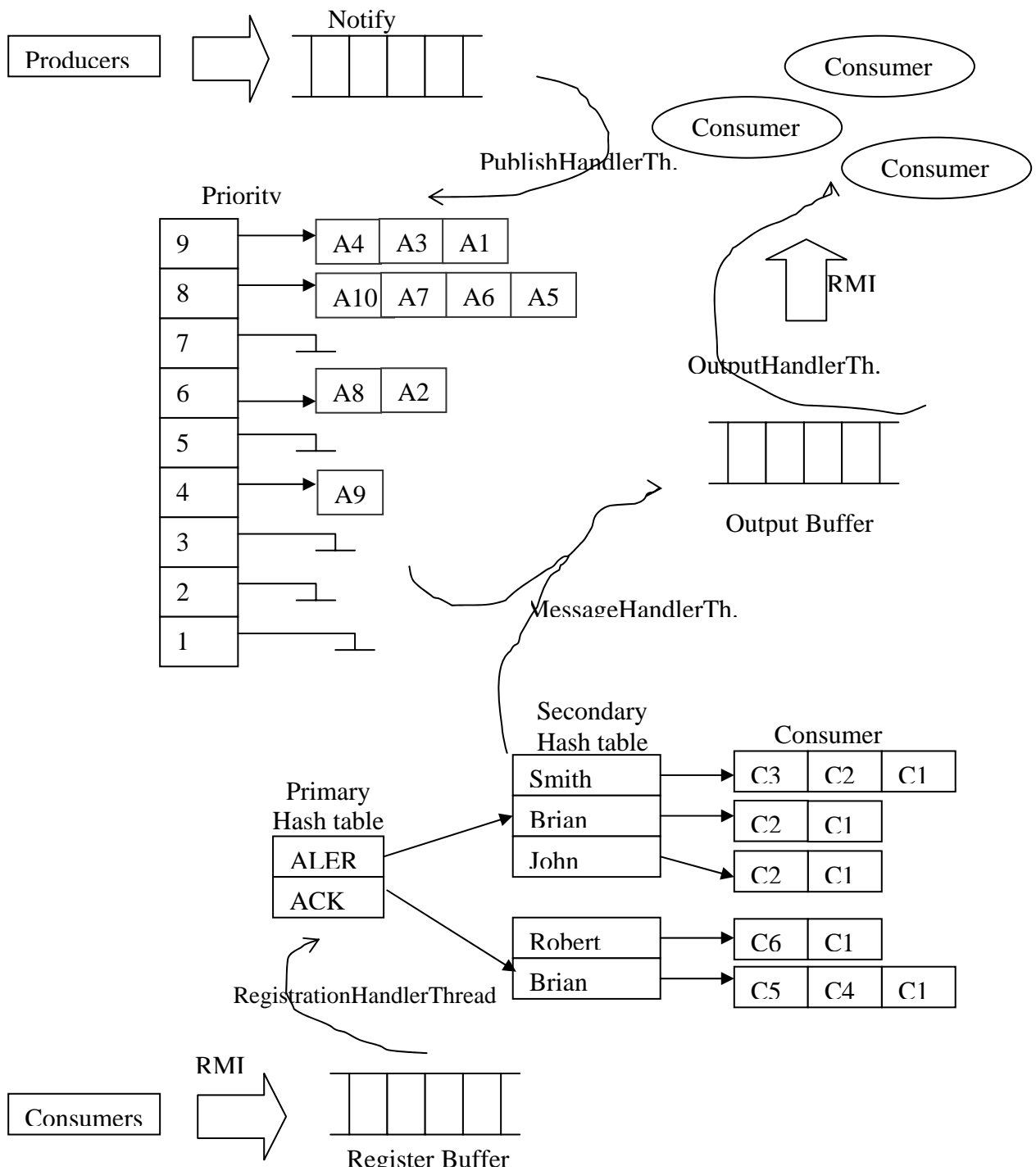


Figure 7. Alert Server Architecture Overview.

5.2. Types of Locks

5.2.1. Mutex

Mutex lock is a synchronization primitive that allows multiple threads to synchronize access to shared data by providing mutual exclusion. The mutex lock has only 2 states: locked and unlocked. Once a thread has acquired the mutex lock on a data structure, other threads attempting to lock the structure will be blocked until it is unlocked. Since mutex allows only one thread to access any data at a given time, it is the most restrictive type of access control. For example, when a mutex is used to synchronize access to a list, the mutex will control the entire list. While the list is being accessed by one thread it is unavailable to all other threads. If most accesses are reads and writes of the existing nodes as opposed to insertions and removes, then a more efficient approach will be to allow nodes to be individually locked.

5.2.2. Read-write

Read-write lock is another synchronization primitive that was designed specifically for situations where shared data is read often by multiple threads/ tasks and rarely written. A read-write lock is similar to a mutex lock except that it allows multiple threads to concurrently acquire the read lock whereas only one writer at a time may acquire a write lock. In the current scenario the *Insert* or *delete* operation on a list will require acquiring the read-write lock in the *write_lock* mode, while the seek (search) of a node will require acquiring the lock in the *read_lock* mode. By using the read-write locks we can have parallel search operations in the Alert Server. The only drawback of using read-write locks is that locking operations take more time than the locking operations on mutexes. Hence locking strategy must be chosen carefully. Read-write locks are justified for the *consumer list* and priority queue data structures in the alerts where inserting and deleting is done only once; thereafter all other operations are search operations on the list to find a particular node. *Read_lock* mode can be used to allow threads to search the list in parallel.

5.2.3. Semaphore

Semaphore is a synchronization primitive that has a value associated with it, which is the number of shared resources regulated by the semaphore. Whenever a thread acquires a semaphore, the semaphore count is decreased by 1. Whenever a thread releases a semaphore, its count is increased by 1. Any thread wanting to acquire the semaphore must wait till its count is greater than 0. Traditionally, semaphore operations have been known as P and V operations. P operation is equivalent to acquiring the semaphore (*sema_wait*). V operation (*sema_post*) is the same as releasing the semaphore. Semaphores are used primarily when there is more than one shared resource that needs to be regulated.

For synchronization of data structures in the Alert Server, mutex locks or semaphores can be used when the operations involved are primarily inserts and deletes

that require exclusive access. For data structures such as the *consumer list*, where a majority of the operations are search operations on the list and updates on individual nodes, read-write locks can be used for locking the list and semaphore or mutex locks can be used for locking individual nodes.

Table 1. Locks used for DIFFERENT DATASTRUCTURES

Data Structures and Characteristics	Locks used with Rationale
<i>Consumer list</i> : list of consumer nodes that are added when a client registers and are deleted when the consumer unregisters with the Alert Server. Whenever the Alert Server has to send an alert to the clients, it scans the list.	Mutex locks are used since operations used are primarily inserts and deletes which happen when a client registers or joins. These operations need an exclusive lock mode that are provided by mutex locks. Using mutex locks is preferred to read-write locks, also because operations on read-write locks have a high overhead.
<i>Alert log file</i> : file that stores the alerts that are published	Mutex locks are used here as file read and write should be mutually exclusive.
<i>Output Queue</i> : queue of alerts and consumers that are added when an alert is to be sent to its consumers.	Mutex locks are used in this case since the only two operations in this case are reading and writing. Nodes are added in the end while the deletion is done from the front.
<i>Secondary Hash table</i> : table of topics with their consumer lists. The table object is added only when there is a new topic.	Read-write lock for locking consumer lists. Write lock provides exclusive access to graph while inserting or deleting nodes in the lists. When accessing list in shared (read) mode, lock hash table is used for managing access to individual lists.

6. CONCLUSIONS AND FUTURE WORK

This thesis presents a messaging system that allows separate client applications to be combined into a flexible and reliable system. It discusses the design and implementation of Alert Server that handles JAVA clients and is also responsible for distributing alerts. It explains the design and implementation of Proxy Server that handles C/C++ clients. It discusses the problems in a messaging environment and the design choices made to solve these problems. It also discusses the various alternatives available and reason behind the choice of a particular alternative.

Alert Server has been implemented using publish/subscribe model where clients can subscribe to three topics TAG, USER and PROFILE. It supports regular expression based subscription in the form of TAG. It also assures delivery of alerts on the basis of priority and maintains transaction logs for a comprehensive audit trail of delivery of alerts, acknowledgements, and receipts to appropriate destinations. It supports persistence of alerts by logging them on to a file-based storage. The clients have the option of sending non-persistent alerts also. A new logging and retrieval mechanism has been implemented. This is more efficient than the traditional way of storing the alerts. This is achieved by storing alerts on the basis of their priorities and storing additional information in the logs. A sweeping algorithm has been designed to sweep the data structures and find the consumers for the alert. A proxy server has also been designed and implemented to handle C/C++ clients. This proxy server is transparent to the clients and its architecture supports a store and forward mechanism. The C/C++ clients are handled in the same way as JAVA clients.

The current implementation of Alert Server is a stand-alone application. This can be integrated with GED/LED so that the clients have an option of either generating events or send/receive alerts. This can be achieved by changing the API of the LED and API of the Alert Server so that the client use that API to decide whether to send an alert or generate an event. The current implementation of Alert Server handles the TAG, USER and PROFILE on one server. This can be changed such that the subscriptions pertaining to USER and PROFILE can be handled by one Alert Server and those pertaining to TAG by another Alert Server. This may be done as Alert Server handles TAG subscriptions slightly differently than the other two. More over this also improves the scalability of Alert Server. The current implementation of Alert Server supports publish/subscribe model. It can be extended to support point-to-point model also. The clients in the current implementation need the IP address of the server in order to communicate. This can be changed to improve availability by using JINI. The programs can then interact spontaneously enabling services to join or leave the network with ease. This allows clients to view and access available services with confidence. The current implementation can also be extended to provide administration utility and also provide encryption mechanism when the alert is sent over a network to provide security.

7. REFERENCES

- [1] Rao, B. R. "Making the Most of Middleware." Data Communications International 24, 12 (September 1995): 89-96.
- [2] The Common Object Request Broker: Architecture and Specification, Version 2.0. Framingham, MA: Object Management Group, 1996. Also available [online] <URL: <http://www.omg.org/>> (1996).
- [3] The Remote Method Invocation Specification.
- [4] Vondrak, C., Message-Oriented Middleware. 1997.
- [5] Object Management, G., {CORBAServices: Common Object Services Specification v1.0}. 1995: John Wiley & Sons Inc. NJ.
- [6] Schmidt, D.C. and S. Vinoski, The OMG Events Service. C++ Report. 1997.
- [7] http://msdn.microsoft.com/library/en-us/cos sdk/htm/pgservices_events_20rp.asp?frame=true, COM+ Events Architecture. 2001.
- [8] Scarlett, S., Monitoring the Behaviour of Distributed Systems, in Cambridge University Computer Laboratory. 1996, University of London: London.
- [9] Dasari, R., Events And Rules For JAVA: Design And Implementation Of A Seamless Approach, in Database Systems R&D Center, CIS Department. 1999, University of Florida: Gainesville.